The 3rd Joint International Conference on Multibody System Dynamics
The 7th Asian Conference on Multibody Dynamics
June 30-July 3, 2014, BEXCO, Busan, Korea

Towards Real-time Multibody Simulations using ARM-based Embedded Systems

Roland Pastorino*#, Francesco Cosco*, Frank Naets*, Javier Cuadrado#, Wim Desmet*

* PMA division
Department of Mechanical Engineering, KU Leuven
Celestijnenlaan 300b, 3001 Leuven, Belgium
roland.pastorino@mech.kuleuven.be
francesco.cosco@mech.kuleuven.be
frank.naets@mech.kuleuven.be
wim.desmet@mech.kuleuven.be

Laboratorio de Ingeniería Mecánica
Universidad de La Coruña
c/. Mendizábal s/n, 15403 Ferrol, Spain
rpastorino@udc.es
javicuad@cdf.udc.es

ABSTRACT

The real-time simulation of multibody models on embedded systems is of particular interest for controllers and observers such as model predictive controllers and state observers, which rely on a dynamic model of the process and are customary executed in electronic control units. This work first identifies the software techniques and tools required to easily write efficient code for multibody models to be simulated on ARM-based embedded systems. Automatic Programming and Source Code Translation are the two techniques that were chosen to generate source code for multibody models in different programming languages. Automatic Programming gives the possibility to generate procedural code from an object-oriented library and Source Code Translation allows to write multibody code in an interpreted language and to translate it automatically to a compiled language for efficiency purposes. Then an implementation of these techniques is proposed. It is based on a `Python` template engine for Source Code Generation and on a model-driven translator for the Source Code Translation. The code is translated from a metalanguage, which is also a Domain Specific Language, to any of the following three programming languages: `Python-Numpy`, `Matlab`, `C++-Armadillo`. Two examples of multibody models were taken from the IFToMM benchmark website: a four-bar linkage with multiple loops and the Andrews' mechanism. The code for these examples has been generated and executed on two ARM-based single-board computers: the `Raspberry Pi Model B` and the `BeagleBone Black`. Using compiled languages, real-time was achieved for the first example and good efficiency was also achieved for the second example despite the low resources and performances of these embedded systems compared with the performances of actual personal computers. This work shows that Automatic Programming and Source Code Translation are valuable techniques to develop multibody models to be used in observers or controllers that are executed in electronic control units.

1 INTRODUCTION

In the Mechatronics industry, the development of control software is divided in three phases [6]:

1. Control software development using a simulation of the plant in a non real-time environment
2. Hardware-In-The-Loop (HIL) simulation and testing in a real-time environment including the target embedded control module
3. Testing and validation of the controller running on the target embedded control module on the real prototype

The models of interest in this research are multibody (MB) models representing the dynamics of mechanical systems. In the first phase, non real-time MB models are commonly used as "plant model" in order to design controllers in a non real-time environment. In the second phase, real-time MB models have been developed during the last 15 years for HIL applications like for example Electronic Control Unit (ECU) testing [11, 19, 26]. In these applications, the MB model, which represents the "plant model", has to be simulated in real-time on a personal computer or workstation. The inputs and outputs of the model are sent/retrieved from the ECU that runs the control software. In the scope of this research, ECU refers to an embedded system in charge of controlling a system with real-time constraints.

Apart from using MB models to help designing and testing controllers, these models can also be used as part of the control software [10, 25]. For example, state observers [29] and model predictive controllers [14] rely on a dynamic model of the process. In these cases, the MB model contained in the controller must be executed on the ECU with the rest of the control software. As ECUs have limited resources and computational performances, the real-time execution of the numerically-intensive MB simulations is more challenging than when using a personal computer. In estimation problems using MB models, a considerable part of the computational cost is due to the evaluation of the model [10, 25]. Therefore, before using MB models for estimation on embedded systems, it is important to evaluate the techniques required for executing MB models in real-time on these platforms as well as the maximum model complexity that can be reached for a particular one.

The continuous improvement of processor performances that took place until recently benefited personal computers as well as embedded systems. During the last decade, thanks to the handset industry, the multiple ARM architectures became the leading processor architectures for embedded systems and hand-held devices because of their reduced cost, heat, power use and their numerous features [8]. They are based on a Reduced Instruction Set Computing (RISC) architecture as opposed to the Complex Instruction Set Computing (CISC) architectures used in Personal Computers (PC). ARM processor performances enabled since the early 2000's the use of modern Operating Systems (OS) on embedded systems, like *embedded GNU/Linux* [20]. Moreover, ARM floating point architecture (VFP) provides hardware support for floating point operations in double-precision floating point arithmetic. As a consequence, ARM-based embedded systems are ideal candidate platforms for running numerically-intensive MB simulations.

This research aims first at identifying the software techniques required to easily program and develop MB models to be executed on ARM-based embedded systems. Then, an implementation of these software techniques is proposed in order to assess the maximum model complexity that can be reached for real-time MB simulations on two particular platforms.

The paper is organized as follows: section 2 focuses on Automatic Programming and Source Code Translation techniques for the generation of minimal procedural source code in several programming languages. Section 3 presents the MB models and the embedded systems used for benchmarking. The settings of the embedded systems for real-time simulations, the benchmark results and performance comparisons are also discussed in this section. Finally, section 4 draws the conclusions. The importance of the tools developed for Automatic Programming and Source Code Translation in the context of the real-time simulation of MB models on ARM-based embedded systems is highlighted.

2 AUTOMATIC PROGRAMMING AND SOURCE CODE TRANSLATION

2.1 Why using Automatic Programming and Source Code Translation for real-time MB models?

Interpreted languages like `Matlab` [31] or `Python` with the `Numpy` library for numerical computing [21] are well suited for code rapid prototyping, algorithm development or data analysis in Scientific Computing. However they are not a good choice for real-time MB simulations [4] because of their low computational efficiency and the lack of determinism of their interpreters (`Matlab engine` or `CPython` interpreter for example). Instead, compiled languages such as `C`, `C++` or `Fortran` using different numerical libraries are commonly employed for efficient MB simulations [11]. When carefully used, compiled languages do not impact negatively the determinism of the simulation. Consequently, when real-time MB simulations are required by the target application, software development is done with a compiled language.

Object-Oriented Programming is a common programming paradigm to build modular, scalable and reusable software in which code and data are grouped into objects. In contrast, Procedural Programming is a programming paradigm based upon procedure calls in which code and data are separated. Both paradigms (one or the other) are used in the design of libraries for the analysis of MB systems [18].

The early choice of the target programming language and the programming paradigm has two main drawbacks:

1. **software development slowdown**: it is difficult to optimally take advantage of interpreted languages

for code rapid prototyping during the software development phase when the chosen programming language is a compiled one (typically for efficiency reasons). One possibility is to write and test new code with interpreted languages and, once ready, to translate it into the target language and incorporate it into the software. These operations are both time-consuming and error-prone. Another possibility is to use interpreted languages and to translate the code to `C/C++` with generic code generators. `MATLAB Coder` is a tool that can generate `C/C++` code from `Matlab`. `Cython` is a programming language that is a superset of `Python` [3]. It allows to convert `Python` code to optimized `C` code by adding static type declarations in the hotspots of the `Python` code. These code translation tools enable important speed-ups. However the generated code is not always sufficiently readable to be analysed and/or modified. On top of that, the programmer has no or few control over the code generation process. This means that with these tools, the generated code is not minimal, the programmer cannot choose the target programming language nor the target library for numerical computing and that important code optimizations can be missed. Due to the limited resources and performance of embedded systems, if some optimizations are not performed on the generated code or if the library providing the most efficient implementations of the required numerical recipes cannot be used, the maximum size of the MB model that can be simulated will be smaller or the MB simulation for a given problem might not be executed in real-time.

2. **Procedural or Object-Oriented Programming**: each programming paradigm has its strengths and weaknesses and the developer can only choose one. Procedural programs are straight-forward implementations of algorithms. This means that they are normally easier to read and analyse than their object-oriented counterpart if they are short. Object-oriented software is more modular, scalable and reusable. It is therefore more appropriate to build large libraries for example. However the higher level of abstraction of object-oriented code comes with some loss of efficiency [7, 22]. This overhead is known as *abstraction penalty*. For this last reason, there is no consensus on the use of Object-Oriented Programming (OOP) or Procedural Programming (PP) in embedded systems [22]. It should also be noted that the performance improvements of embedded systems in the last years mitigates this problem.

At this point it should be clear that the best scenario for the real-time simulations of MB models on embedded systems is to develop the code using interpreted languages, to organize data and methods in an object-oriented library, to write the code to be executed in a compiled language with the chosen numerical library and using procedures instead of objects. OOP or PP considered separately fail to comply with this best scenario. Automatic Programming (AP) and Source Code Translation (SCT) are the two software techniques proposed in this research to take advantage of the strengths of both OOP and PP in the same software.

2.2 Automatic Programming

AP is a programming paradigm in which a program writes another program in order to let the developer write code at a higher level of abstraction. Typically, a source code generator gathers information from objects and writes a procedural program. AP is a technique used currently in the fields of embedded systems [22], Mechatronics [33] and in the Automotive field [12]. The code that is produced by the source code generator can be efficiently compiled to machine code with modern compilers. It is normally difficult to further improve by hand this code. AP is an attractive technique as it allows to increase the level of abstraction without compromising code efficiency.

In the scope of real-time MB applications, AP is a valuable tool. It enables the high-level definition of MB models. The definition files of MB models can be read and interpreted by an object-oriented library which contains both data and methods. Finally a source code generator can generate minimal procedural code for a particular model. In few words, AP for real-time MB applications combines the following advantages:

- high-level definition of MB models
- generation of procedural minimal source code that avoids software overheads
- reduced compilation time as only the minimal code has to be compiled. For embedded systems, this means that the source code can be cross-compiled from the development PC to the embedded system (i.e. cross build) or even built natively on the embedded system if a compiler is available (i.e. native build).

- generated source code for MB models can be exported to other libraries (e.g. library for Estimation)

2.3 Source Code Translation

A translator is a program that maps input constructs to output constructs [24]. It allows to implement new Domain Specific Languages (DSL) and programming languages by translating them to existing languages. As mentioned in section 2.1, the use of a single programming language (like `C` or `C++`) to develop a software limits the ability of developers in taking advantage of other languages (like `Matlab` or `Python`) for the development of new algorithms. However, in the field of MB analysis, it can be noted that the mathematical operations commonly used are identical for all the languages and numerical libraries except for their syntax. Subsequently it is possible to write the different functions of a program in one language and translate them to other languages. The primary language can be qualified as a metalanguage and a DSL (in the remaining part of this paper, it will be referred only as metalanguage for the sake of simplicity). The other languages are target or object languages. In short, Source Code Translation has the following advantages:

- code rapid prototyping in an interpreted language instead of a compiled language to extend the library
- choice of the most suitable programming language for the target application
- choice of the most suitable library for numerical computing for the target application
- maximum efficiency can be reached using compiled programming languages and efficient numerical libraries
- genuine comparison of the accuracy and efficiency of different implementations of a MB model (language, numerical library, MB formulation, etc) can be performed

2.4 Software implementation for Automatic Programming and Source Code Translation

AP and SCT can be done in many ways. The implementation proposed in this research is presented hereafter. A library based on the general-purpose, high-level, interpreted programming language `Python 3` has been built. This programming language has been chosen because it is cross-platform, its standard library provides useful tools like the `ast` module (discussed later) and it is readily available for all the `GNU/Linux` distributions. `GNU/Linux` is the main target OS for embedded systems in this research. Three sets of target programming language and numerical library have been selected:

- `Python-Numpy` for code rapid prototyping, easy debugging and cross-platform simulations
- `Matlab` for the same reasons as above and for its widespread use in numerical simulations
- `C++-Armadillo` for computational efficiency [27]. `Armadillo` is based on a delayed evaluation approach during compile time to combine several operations into one in order to reduce or eliminate the need for temporaries. Only dense matrices have been used.

Figure 1 shows a scheme of the structure of the software. The user provides a file containing the definition of a MB model. Then the object-oriented library core parses it and encapsulates the information in the corresponding objects. After that, source code is generated using the `Python` template engine `Jinja2` and `print` statements. Then, the resulting generated code is translated from the metalanguage to the target programming language. A standalone file containing the MB model programmed in the target language and target numerical library is now available to the user. The source code generation (SCG) and the SCT are both discussed hereafter in greater detail.

Source Code Generation via template engine - A template engine is a piece of software designed to combine one or mores templates with a data model to produce one or more result documents. The `Python` template engine `Jinja2` has been selected to generate the part of the source code that is too specific to a particular language to be written in metalanguage. This represents only a minimal part of the generated source code.

Source Code Translation via metalanguage - As mentioned in section 2.3, a metalanguage is a useful tool to write MB code once and for all the target languages. Instead of creating a new metalanguage from scratch, the metalanguage used in this research is based on an extension of the grammar of `Python`

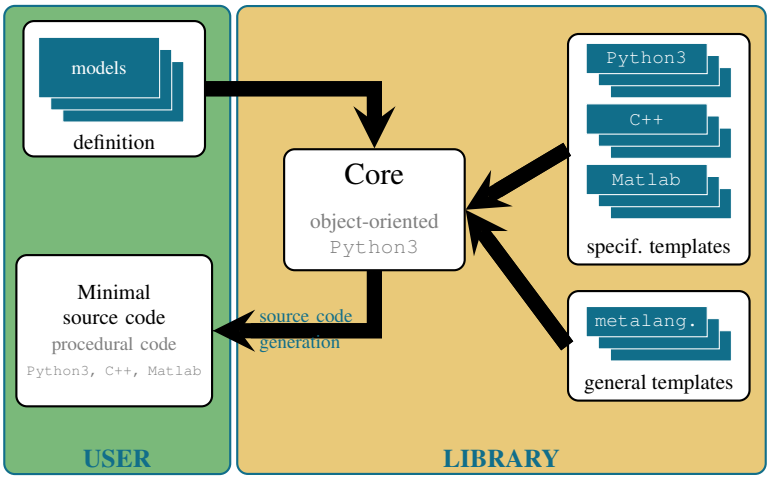
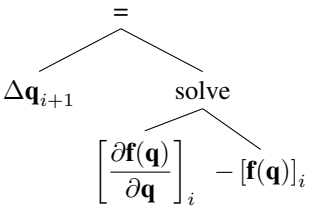


Figure 1: Source Code Generation via templates and metalanguage.

3. This has the advantage of allowing the developer to program both the library core and the MB code contained in the library in one single language: `Python 3`. The implemented translator is a model-driven translator [24] meaning that first an Abstract Syntax Tree (AST) for each metalanguage piece of code is created. Then every AST is walked and output is generated with `print` statements based on the grammar of the target language. As an example of this process, equation (1) shows the Newton-Raphson method used by the implicit integrator mentioned in section 3.1. The AST corresponding to the solution of this linear system is shown in figure 2a. It was automatically generated by the `ast` module of the `Python` standard library, Every node of this AST has to be translated following the rules defined in the grammar of the target language. Figure 2b shows the translation rule of the `solve` function for all the target languages.

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]_i \Delta\mathbf{q}_{i+1} = -\left[\mathbf{f}(\mathbf{q})\right]_i \tag{1}$$



(a) `Metalanguage AST`

Language	Function `solve`
`Python-Numpy`	`numpy.linalg.solve`
`Matlab`	\
`C++-Armadillo`	`solve`

(b) Grammar rules for the `solve` function

Figure 2: Example of AST and grammar rule for Source Code Translation

3 BENCHMARK OF MB MODELS ON DIFFERENT PLATFORMS

3.1 IFToMM benchmark examples

Two MB models were taken from the *IFToMM* benchmark website [30] and their source code was generated using the different available combinations of language and numerical library. These models are briefly presented hereafter.

N-loop four-bar linkage: this 2D one degree-of-freedom mechanism is an assembly of N-loop four-bar linkages, as shown in figure 3a. It is made of thin rods of 1 m length with a uniformly distributed mass of 1 kg and moves under the effect of gravity. This mechanism undergoes singular configurations. When

the mechanism reaches the horizontal position, the number of degrees of freedom instantaneously increases from 1 to N+1. Some formulations have problems to overcome this situation. The simulation time is 10 s and the fixed integration time step 10 ms. The complete description of the problem including the initial conditions can be found in the *IFToMM* benchmark website. It has been modelled using planar natural coordinates leading to *2N+2* variables and *2N+1* constraints associated to the constant distance condition of each rod. The equations of motion are given by the index-3 augmented Lagrangian formulation [13] shown in equation (2).

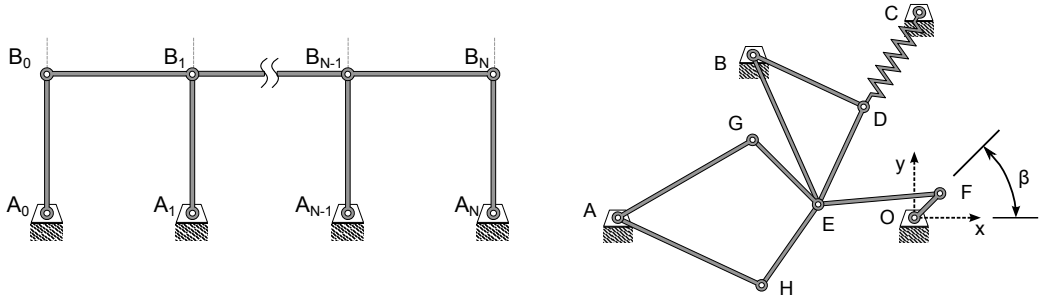
$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_\mathbf{q}^T \boldsymbol{\alpha}\mathbf{\Phi} + \mathbf{\Phi}_\mathbf{q}^T \boldsymbol{\lambda}^* = \mathbf{Q} \\ &\boldsymbol{\lambda}_{i+1}^\star = \boldsymbol{\lambda}_i^\star + \alpha \mathbf{\Phi}_{i+1} \qquad i = 0, 1, 2... \end{aligned} \tag{2}$$

where $\mathbf{M}$ is the mass matrix (constant for this example), $\ddot{\mathbf{q}}$ are the accelerations, $\mathbf{\Phi_q}$ is the Jacobian matrix of the constraints, α is the penalty factor, $\mathbf{\Phi}$ is the vector of constraints, $\boldsymbol{\lambda}^*$ are the Lagrange multipliers and $\mathbf{Q}$ is the vector of applied and velocity dependent inertia forces. The Lagrange multipliers for each time-step are obtained from an iterative process where the value of $\boldsymbol{\lambda}_0^*$ is equal to the $\boldsymbol{\lambda}^\star$ obtained in the previous time-step. Regarding the integration scheme, the implicit single-step trapezoidal rule has been selected. The difference equations of this integrator have been introduced into the equations of motion and then solved using a Newton-Raphson scheme as shown in [15]. Finally cleaned velocities $\dot{\mathbf{q}}$ and cleaned accelerations $\ddot{\mathbf{q}}$ were obtained using mass-orthogonal projections [9].

Andrew's mechanism or seven-bar linkage: this 2D mechanism is composed of seven bodies connected by joints without friction, as shown in figure 3b. The numerical constants were taken from [16, 28]. This example has a very small time scale thus requiring small time steps. Mixed coordinates [13] were used leading to 11 variables and 10 constraints associated to the constant distance of each body and to the angle β. The equations of motion are given by the index-1 augmented Lagrangian formulation [13] shown in equation (3).

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_\mathbf{q}^T \boldsymbol{\alpha}\ddot{\mathbf{\Phi}} + \mathbf{\Phi}_\mathbf{q}^T \boldsymbol{\lambda}^* = \mathbf{Q} \\ &\boldsymbol{\lambda}_{i+1}^\star = \boldsymbol{\lambda}_i^\star + \alpha \ddot{\mathbf{\Phi}}_{i+1} \qquad i = 0, 1, 2... \end{aligned} \tag{3}$$

Regarding the integration scheme, the implicit single-step trapezoidal rule has been selected. The difference equations of this integrator have been introduced into the equations of motion and then solved using a Newton-Raphson scheme as shown in [9]. Finally cleaned positions $\mathbf{q}$ and cleaned velocities $\dot{\mathbf{q}}$ were obtained using mass-orthogonal projections [9].



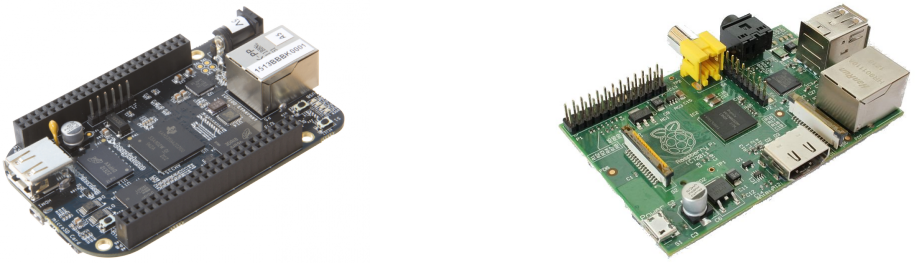
(a) `N-loop four-bar linkage` (b) `Andrews' mechanism`

Figure 3: Benchmark MB examples

3.2 ARM-based embedded systems

Within the innumerable number of embedded systems available on the market, ARM-based single-board computers (SBC) are good candidates to run the numerically-intensive MB simulations [8]. There exists a multitude of them ranging from boards with a single-core processor and 64 MB of RAM up to boards with an eight-core processor and 3 GB of RAM. Usual Operating Systems (OS) are `Android` or a distribution of `GNU/Linux` such as `Ubuntu` or `Fedora`.

The two ARM-based embedded systems that were selected in this research are the `Raspberry Pi Model B` (RPI) [17] and the `BeagleBone Black` (BBB) [2]. These single-core processor SBCs run a distribution of `GNU/Linux`. Some specifications of interest for both SBCs are summarized in Table 1.



(a) `BeagleBone Black` (b) `Raspberry Pi`

Figure 4: tested ARM-based embedded systems

Table 1: main specifications of the RPI and the BBB.

	Processor	Freq.	L1d cache	L1i cache	L2 cache	SDRAM	OS
BBB	ARM Cortex A8	1 GHZ	32 KB	32 KB	256 KB	512MB	Ubuntu 13.10 kernel 3.8
RPI	ARM 1176JZF-S	700 MHz	32 KB	32 KB	128 KB	512MB	Raspbian kernel 3.10

3.3 Settings for real-time simulations

Section 2 dealt with software techniques for generating efficient code for a MB model. This is a necessary but not sufficient condition to run MB models on embedded systems in real-time. Numerous definitions of *real-time system* can be found in the literature [22]. The one taken into account in this research is: "A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness". It is worth mentioning that in the field of MB systems, the word `real-time` is mainly used to refer to efficient computational methods instead of being related to the determinism of the system or simulation.

As mentioned in the previous section, the OS of both embedded systems is a distribution of `GNU/Linux`. For this OS, two approaches exist to improve its real-time performances [5, 32]. The first approach is to patch the kernel in order to make it more preemptible thus improving interrupt latency, context switching time, scheduling latency and providing a finer timer granularity [1]. The second approach is to use a sub-kernel. In this case, a small real-time kernel runs the Linux kernel as a low priority task. Both approaches were followed for the BBB (due to the availability of the `CONFIG_PREEMPT_RT` patch for Ubuntu 13.10) and only the second approach was followed for the RPI. Then, minimal source code for the MB models was generated using `C++-Armadillo`. As described in [23], the highest real-time priority was given to the process and all pages were locked into memory in order to avoid page faults. Finally, the dynamic CPU frequency scaling was disabled by using the utility `cpufreq` to set the CPU frequency to its maximum (as shown in table 1).

Following the test proposed in [23], the `nanosleep` benchmark was executed on the BBB to assess the jitter of this SBC. The jitter dropped from a few milliseconds for the vanilla Linux kernel to about 50

microseconds for the patched kernel. It is worth mentioning that if a vanilla Linux kernel is used for executing a real-time MB simulation with for example a 5 ms time step, the jitter mentioned above and the integration time step will in the same order of magnitude. Whether this is an acceptable situation or not depends on the requirements of the application.

3.4 Results and comparison between ARM-based embedded systems and personal computer

Simulations using both aforementioned MB models have been carried out using the BBB and the RPI. Every simulation has been executed several times and only the fastest execution time has been taken into account. The dispersion of the execution times was much lower when using the `CONFIG_PREEMPT_RT` patch than with the vanilla Linux kernel. The only load on the processor of the SBCs was the MB simulation and an `ssh` connection.

Figure 5 shows, on a logarithmic scale, the real-time factor for the N-loop four-bar linkage in function of the number of variables. This number was increased by changing the number of loops of the mechanism. The real-time factor is the ratio of the simulation time and the execution time; thus real-time simulations have a real-time factor superior or equal to 1. The solid lines correspond to the `C++-Armadillo` simulations. The simulation times for the BBB and the RPI are very similar despite the different ARM architectures and the different CPU frequencies. For both of them, the maximum number of variables for real-time simulations is 40. To be considered safe, a real-time system should have a CPU utilization factor between 51% and 68% [22]. This implies that the number of variables has to be lower than 40 for having a safe real-time system. Then, the dashed lines correspond to the `Python-Numpy` simulations. They are approximately 60 times slower than the `C++` ones for the BBB and 150 times for the RPI. As expected, `Python-Numpy` is not an option for real-time MB simulations on embedded systems but is a good tool for code rapid development (see section 2). The origin of the important difference between both SBCs for `Python-Numpy` simulations is probably due to the different `Python` interpreter versions. The BBB with `Ubuntu 13.10` uses `CPython 3.3.2` while the RPI with `Raspbian` uses `CPython 3.2.3`.

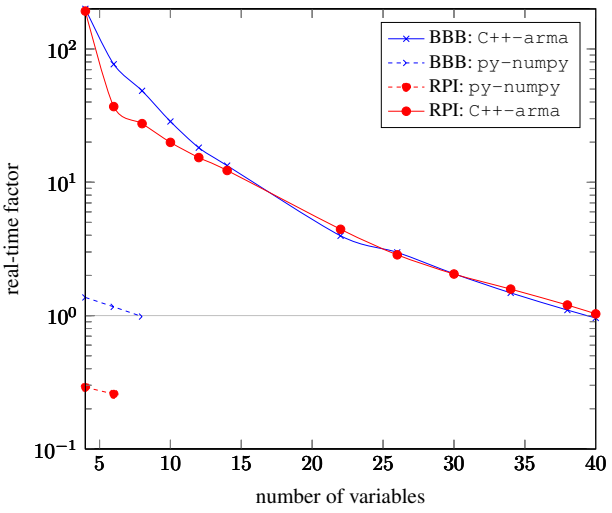


Figure 5: Real-time factor for the N-loop four-bar linkage on the RPI and the BBB

Table 2 presents the execution times for the Andrews' mechanism. The simulation time was 0.03 s and the time step size was 0.01 ms. Real-time was not reached due to the very small time step imposed by the nature of the problem. The difference in execution time between `C++-Armadillo` and `Python-Numpy` are similar to the one of the other example.

In order to evaluate the impact of using ARM-based SBCs instead of CISC-based PCs, a simulation of the N-loop four-bar linkage with 2 loops was executed on a personal computer running `Ubuntu 13.10` on an `Intel Core i7-3740QM` processor. The simulation settings were the same as the one mentioned in section 3.3. The CPU frequency was fixed at 2.7 GHz. Table 3 shows the corresponding execution times. For `C++-Armadillo` the simulations on the SBCs were between 15 and 60 times slower while for

Table 2: Simulation time (s) of Andrews' mechanism on ARM embedded systems.

	BBB	RPI
`Python-Numpy`	29.47	94.9176
`C++-Armadillo`	1.3026	1.4364

`Python-Numpy` they were between 30 and 60 times slower than for the PC. These numbers give an idea of the slowdown that can be expected for MB code when using ARM-based SBCs instead of CISC-based computers.

Table 3: Simulation time (s) of a double four-bar linkage on ARM and CISC processors.

	BBB	RPI	`i7-3740QM`
`Python-Numpy`	8.2430	34.1409	0.5574
`C++-Armadillo`	0.1278	0.2706	0.0045

4 CONCLUSIONS

This research has first identified the software techniques required to easily program and develop MB models to be executed on ARM-based embedded systems. Automatic Programming is the technique that allowed to generate minimal source code from the high-level definition of a MB model. It gives the possibility to decouple the use of object-oriented programming for the structure of the software and the use of procedural programming for the generated MB code. Source Code Translation is the technique used to translate the generated minimal source code to a target language, using the corresponding grammar. Developers can therefore generate MB code in their preferred target language, improve the standalone generated file and then include these modifications back into the MB library.

In the second phase of this research, an implementation of the aforementioned techniques has been proposed. Automatic Programming has been done using a `Python` template engine and `print` statements. The developed MB library is object-oriented and programmed in `Python 3` while the generated code is procedural and programmed in metalanguage. Source Code Translation has been implemented with a model-driven translator. An abstract syntax tree is first built from the generated code and then walked to generate the translated code based on the grammar of the target language. Target languages for both code rapid prototyping (`Python-Numpy` and `Matlab`) and computational efficiency (`C++-Armadillo`) were implemented.

Finally, the code for two MB examples taken from the IFToMM benchmark website has been generated using the developed library. Every generated MB code was executed on two ARM-based single-board computers running a distribution of Embedded GNU/Linux. The use of the `CONFIG_PREEMPT_RT` patch and sub-kernels for improving the determinism of the Linux kernel were taken into account. These benchmarks have demonstrated that the simulations in `C++-Armadillo` can be executed faster than real-time on the proposed embedded systems. The maximum number of variables of a MB model for real-time simulations on these embedded systems depends on multiple factors: integrator, time step size, CPU utilization factor, multibody formulation, numerical library, etc.

Regarding future work, this research has shown that executing real-time rigid multibody simulations on ARM-based embedded systems is feasible and realistic. As a consequence, a possible extension of this work could be to use rigid multibody models for Estimation and/or Control of mechanical systems on ARM-based embedded systems.

5 ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of the European Commission through the Marie Curie IAPP project "Interactive" (Innovative Concept Modelling Techniques for Multi-Attribute Optimization of Active Vehicles), with contract number 285808 (http://www.fp7interactive.eu), the IWT Flanders and ITEA2 through the MODRIO project, the Research Fund KU Leuven. This work benefits also from the Belgian Programme on Interuniversity Attraction Poles, initiated by the Belgian Federal Science Policy Office (DYSCO). The research of Roland Pastorino is also funded by grant GCP2013/056 of the Galician Government. The research of Frank Naets is funded by a postdoctoral fellowship of the Fund for Scientific Research, Flanders (F.W.O).

REFERENCES

[1] J. Altenberg. "Using the Realtime Preemption Patch on ARM CPUs". In: *Eleventh Real-Time Linux Workshop*. Dresden, Germany, Sept. 2009.

[2] S. F. Barrett and J. Kridner. *Bad to the Bone: Crafting Electronic Systems with BeagleBone and BeagleBone Black (Synthesis Lectures on Digital Circuits and Systems)*. Ed. by Morgan & Claypool Publishers. Morgan Claypool Publishers, May 2013, p. 424.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. "Cython: The Best of Both Worlds". In: *Computing in Science and Engineering* 13.2 (2011), pp. 31–39. ISSN: 1521-9615.

[4] J. Bélanger, P. Venne, and J. Paquin. *The What, Where and Why of Real-Time Simulation*. Tech. rep. OPAL-RT, Oct. 2010.

[5] J. H. Brown and B. Martin. "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications". In: *Twelfth Real-Time Linux Workshop*. 16. Nairobi, Kenya, Oct. 2010.

[6] S. Cetinkunt, S.-i. Nakajima, B. Nelson, and S. Haggag. "Embedded-Model-Based Control". In: *Journal of Control Science and Engineering* 2013.237897 (2013), p. 2.

[7] A. Chatzigeorgiou. "Performance and power evaluation of C++ object-oriented programming in embedded processors". In: *Information & Software Technology* 45.4 (2003), pp. 195–201.

[8] S. Chon and A. Valenzuela. *Getting Started on TI ARM embedded processor development – The Basics*. Tech. rep. Texas Instruments, 2013.

[9] J. Cuadrado, J. Cardenal, and E. Bayo. "Modeling and Solution Methods for Efficient Real-Time Simulation of Multibody Dynamics". English. In: *Multibody System Dynamics* 1.3 (1997), pp. 259–280. ISSN: 1384-5640. URL: http://dx.doi.org/10.1023/A:1009754006096.

[10] J. Cuadrado, D. Dopico, J. A. Pérez, and R. Pastorino. "Automotive Observers Based on Multibody Models and the extended Kalman filter". In: *Multibody System Dynamics* 27.1 (2011), pp. 3–19.

[11] A. Eichberger. *Generating multibody real-time models for hardware-in-the-loop applications*. 2002.

[12] B. Fijalkowski. "Model-Based Design with Production Code Generation for SBW AWS Conversion Mechatronic Control System Development". English. In: *Automotive Mechatronics: Operational and Practical Issues*. Vol. 52. Intelligent Systems, Control and Automation: Science and Engineering. Springer Netherlands, 2011, pp. 153–160. ISBN: 978-94-007-1182-2. URL: http://dx.doi.org/10.1007/978-94-007-1183-9_7.

[13] J. García de Jalón and E. Bayo. *Kinematic and Dynamic Simulation of Multibody Systems: The Real–Time challenge*. Ed. by Springer–Verlag. Springer–Verlag, 1994.

[14] C. E. Garcia, D. M. Prett, and M. Morari. "Model predictive control: Theory and practice – A survey". In: *Automatica* 25.3 (1989), pp. 335–348. ISSN: 0005-1098. URL: http://www.sciencedirect.com/science/article/pii/0005109889900022.

[15] M. González, F. González, D. Dopico, and A. Luaces. "On the effect of linear algebra implementations in real-time multibody system dynamics". English. In: *Computational Mechanics* 41.4 (2008), pp. 607–615. ISSN: 0178-7675. URL: http://dx.doi.org/10.1007/s00466-007-0218-2.

[16] E. Hairer, S. P. Norsett, and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Ed. by Springer. 2nd Revised Edition. Berlin: Springer, 1991.

[17] G. Halfacree and E. Upton. *Raspberry Pi User Guide*. Ed. by Wiley. 1st ed. Wiley, 2012, p. 262.

[18] A. Kecskemethy and M. Hiller. "An object-oriented approach for an effective formulation of multibody dynamics". In: *Computer Methods in Applied Mechanics and Engineering* 115.3-4 (1994), pp. 287–314. ISSN: 0045-7825. URL: http://www.sciencedirect.com/science/article/pii/0045782594900647.

[19] S. S. Kim, H. K. Jung, J. S. Shim, and C. W. Kim. “Development of vehicle dynamics model for real–time electronic control unit evaluation system using kinematic and compliance test data”. In: *International Journal of Automotive Technology* 6.6 (2005), pp. 599–605.

[20] H. Kingman. “The History of Embedded Linux & Best Practices for Getting Started”. In: *Linux Foundation Training Publication* (2013).

[21] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Ed. by S. B. Heidelberg. Vol. 6. Springer Berlin Heidelberg, 2011.

[22] P. A. Laplante and S. J. Ovaska. *Real-Time Systems Design and Analysis*. Ed. by I. John Wiley & Sons. John Wiley & Sons, Inc., 2012.

[23] P. McKenney. “’Real Time’ vs. ’Real Fast’: How to Choose?” In: *Proceedings of the Ottawa Linux Symposium*. 2008.

[24] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Ed. by P. Bookshelf. 1st ed. Pragmatic Bookshelf, Jan. 2010, p. 374.

[25] R. Pastorino, D. Richiedei, J. Cuadrado, and A. Trevisani. “State estimation using multibody models and non-linear Kalman filters”. In: *International Journal of Non-Linear Mechanics* 53 (2013), pp. 83–90. ISSN: 0020-7462.

[26] W. Rulka and E. Pankiewicz. “MBS approach to generate equations of motions for HiL-Simulations in vehicle system dynamics”. In: *Multibody System Dynamics* 14.3–4 (Dec. 2005), pp. 367–386.

[27] C. Sanderson. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Tech. rep. NICTA, Sept. 2010.

[28] W. Schiehlen. *Multibody Systems Handbook*. Ed. by S. B. Heidelberg. Springer Berlin Heidelberg, 1990.

[29] D. Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Ed. by Wiley-Interscience. 1st ed. Wiley-Interscience, June 2006.

[30] *The IFToMM benchmark website*. http://iftomm-multibody.org/benchmark/. Online; accessed March-2014. 2014.

[31] *The MathWorks website*. http://www.mathworks.com. Online; accessed March-2014. 2014.

[32] N. Vun, H. F. Hor, and J. W. Chao. “Real-Time Enhancements for Embedded Linux”. In: *14th IEEE International Conference on Parallel and Distributed Systems*. Dec. 2008, pp. 737–740.

[33] M. A. Wehrmeister, E. P. de Freitas, A. P. D. Binotto, and C. E. Pereira. “Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems”. In: *Mechatronics* (2014), pages. ISSN: 0957-4158. URL: http://www.sciencedirect.com/science/article/pii/S0957415813002420.